JH_task4

December 15, 2022

1 Assignment 4 - Solving a finite element system

This assignment makes up 30% of the overall marks for the course. The deadline for submitting this assignment is **5pm on Thursday 15 December 2022**.

Coursework is to be submitted using the link on Moodle. You should submit a single pdf file containing your code, the output when you run your code, and your answers to any text questions included in the assessment. The easiest ways to create this file are:

- Write your code and answers in a Jupyter notebook, then select File -> Download as -> PDF via LaTeX (.pdf).
- Write your code and answers on Google Colab, then select File -> Print, and print it as a pdf.

Tasks you are required to carry out and questions you are required to answer are shown in bold below.

[1]:
```
from scipy.sparse import coo_matrix
import numpy as np
from time import perf_counter

%matplotlib inline
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
```

1.1 The assignment

1.1.1 Mathematical background

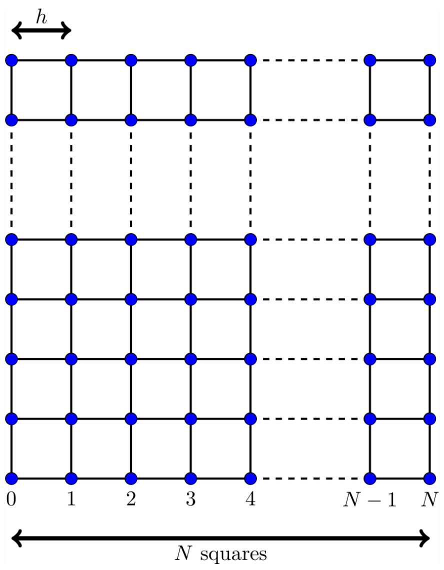
In this assignment, we are going to solve a Helmholtz wave problem:

$$-\Delta u - k^2 u = 0 \text{ in } \Omega$$
$$u = g \text{ on the boundary of } \Omega$$

As our domain we will use the unit square, ie $\Omega = [0,1]^2$. In this assignment, we will use $k = 5$ and

$$g(x,y) = \begin{cases} \sin(4y) & \text{if } x = 0, \\ \sin(3x) & \text{if } y = 0, \\ \sin(3+4y) & \text{if } x = 1, \\ \sin(3x+4) & \text{if } y = 1. \end{cases}$$

The finite element method is a method that can approximately solve problems like this. We first split the square $[0,1]^2$ into a mesh of N squares by N squares (or $N+1$ points by $N+1$ points - note that there are N squares along each side, but $N+1$ points along each side (watch out for off-by-one errors)):



As shown in the diagram, we let $h = 1/N$.

The (degree 1) finite element method looks for an approximate solution by placing an unknown value/variable at each point, and approximating the solution as some linear combination of the functions 1, x, y and xy inside each square. Re-writing the problem as an integral equation (and doing a bit of algebra) allows us to turn the problem into the matrix vector problem

$$\mathrm{A}\mathbf{x} = \mathbf{b}.$$

(We do not need to go into details of how this method is derived, but if you're curious, the first chapter of *Numerical Solution of Partial Differential Equations by the Finite Element Method* by Claes Johnson gives a good introduction to this method.)

Let $\mathbf{p}_0$, $\mathbf{p}_1$, ..., $\mathbf{p}_{(N-1)^2-1}$ be the points in our mesh that are not on the boundary (in some order). Let x_0, x_1, ..., $x_{(N-1)^2-1}$ be the values/variables at the points (these are the entries of the unknown vector $\mathbf{x}$).

A is an $(N-1)^2$ by $(N-1)^2$ matrix. $\mathbf{b}$ is a vector with $(N-1)^2$ entries. The entries $a_{i,j}$ of the matrix A are given by

$$a_{i,j} = \begin{cases} \dfrac{24 - 4h^2k^2}{9} & \text{if } i = j \\ \dfrac{-3 - h^2k^2}{9} & \text{if } \mathbf{p}_i \text{ and } \mathbf{p}_j \text{ are horizontally or vertically adjacent} \\ \dfrac{-12 - h^2k^2}{36} & \text{if } \mathbf{p}_i \text{ and } \mathbf{p}_j \text{ are diagonally adjacent} \\ 0 & \text{otherwise} \end{cases}$$

The entries b_j of the vector $\mathbf{b}$ are given by

$$b_j = \begin{cases} \dfrac{12 + h^2k^2}{36}\left(g(0,0) + g(2h,0) + g(0,2h)\right) + \dfrac{3 + h^2k^2}{9}\left(g(h,0) + g(0,h)\right) & \text{if } \mathbf{p}_j = (h,h) \\ \dfrac{12 + h^2k^2}{36}\left(g(1,0) + g(1,2h) + g(1-2h,0)\right) + \dfrac{3 + h^2k^2}{9}\left(g(1-h,0) + g(1,h)\right) & \text{if } \mathbf{p}_j = (1-h,h) \\ \dfrac{12 + h^2k^2}{36}\left(g(0,1) + g(2h,1) + g(0,1-2h)\right) + \dfrac{3 + h^2k^2}{9}\left(g(h,1) + g(0,1-h)\right) & \text{if } \mathbf{p}_j = (h,1-h) \\ \dfrac{12 + h^2k^2}{36}\left(g(1,1) + g(1-2h,1) + g(1,1-2h)\right) + \dfrac{3 + h^2k^2}{9}\left(g(1-h,1) + g(1,1-h)\right) & \text{if } \mathbf{p}_j = (1-h,1- \\ \\ \dfrac{12 + h^2k^2}{36}\big(g(0,c_j+h) + g(0,c_j-h)\big) + \dfrac{3 + h^2k^2}{9}g(0,c_j) & \text{if } \mathbf{p}_j = (h,c_j)\text{, with} \\ \dfrac{12 + h^2k^2}{36}\big(g(1,c_j+h) + g(1,c_j-h)\big) + \dfrac{3 + h^2k^2}{9}g(1,c_j) & \text{if } \mathbf{p}_j = (1-h,c_j)\text{,} \\ \dfrac{12 + h^2k^2}{36}\big(g(c_j+h,0) + g(c_j-h,0)\big) + \dfrac{3 + h^2k^2}{9}g(c_j,0) & \text{if } \mathbf{p}_j = (c_j,h)\text{, with} \\ \dfrac{12 + h^2k^2}{36}\big(g(c_j+h,1) + g(c_j-h,1)\big) + \dfrac{3 + h^2k^2}{9}g(c_j,1) & \text{if } \mathbf{p}_j = (c_j,1-h)\text{,} \\ 0 & \text{otherwise} \end{cases}$$

You could alternatively write this as

$$b_j = \frac{12 + h^2k^2}{36}\big(\text{sum of evaluations of } g \text{ at all points on the boundary that are diagonally adjacent to } \mathbf{p}_j\big) +$$

$$\frac{3 + h^2k^2}{9}\big(\text{sum of evaluations of } g \text{ at all points on the boundary that are horizontally or vertically adjacent to } \mathbf{p}_j$$

For example (using k and g as given above) when $N = 2$,

$$\mathrm{A} = \begin{pmatrix} -0.11111111 \end{pmatrix}.$$

For $N = 2$, the definition of $\mathbf{b}$ is different to above, as the point at $(1/2, 1/2)$ is adjacent to all three sides and so the conditions above are all true at once. The alternate value of $\mathbf{b}$ used in this case is not important, as we will later take $N > 2$.

As as second example, when $N = 3$,

$$A = \begin{pmatrix} 1.43209877 & -0.64197531 & -0.64197531 & -0.41049383 \\ -0.64197531 & 1.43209877 & -0.41049383 & -0.64197531 \\ -0.64197531 & -0.41049383 & 1.43209877 & -0.64197531 \\ -0.41049383 & -0.64197531 & -0.64197531 & 1.43209877 \end{pmatrix},$$

$$b = \begin{pmatrix} 1.72513230 \\ 0.15334285 \\ -0.34843455 \\ -1.05586511 \end{pmatrix}.$$

In this second example, I have numbered the points not on the boundary like this:

$$\begin{matrix} 2 & 3 \\ 0 & 1 \end{matrix}$$

1.1.2 Part 1: creating the matrix and vector

Write a function that takes N as an input and returns the matrix A and the vector $\mathbf{b}$. The matrix should be stored using an appropriate sparse format - you may use Scipy for this, and do not need to implement your own format.

[2]:
```
# A and b for N=2
A_2 = np.array([
    [-0.11111111111111116],
])
b_2 = np.array([0.2699980311833446])

# A and b for N=3
A_3 = np.array([
    [1.4320987654320987, -0.6419753086419753, -0.6419753086419753, -0.
 ↪4104938271604938],
    [-0.6419753086419753, 1.4320987654320987, -0.4104938271604938, -0.
 ↪6419753086419753],
    [-0.6419753086419753, -0.4104938271604938, 1.4320987654320987, -0.
 ↪6419753086419753],
    [-0.4104938271604938, -0.6419753086419753, -0.6419753086419753, 1.
 ↪4320987654320987],
])
b_3 = np.array([1.7251323007221917, 0.15334285313223067, -0.34843455260733003,␣
 ↪-1.0558651156722307])

# A and b for N=4
A_4 = np.array([
    [1.9722222222222222, -0.5069444444444444, 0.0, -0.5069444444444444, -0.
 ↪3767361111111111, 0.0, 0.0, 0.0, 0.0],
    [-0.5069444444444444, 1.9722222222222222, -0.5069444444444444, -0.
 ↪3767361111111111, -0.5069444444444444, -0.3767361111111111, 0.0, 0.0, 0.0],
    [0.0, -0.5069444444444444, 1.972222222222222, 0.0, -0.3767361111111111, -0.
↪5069444444444444, 0.0, 0.0, 0.0],
    [-0.5069444444444444, -0.3767361111111111, 0.0, 1.972222222222222, -0.
↪5069444444444444, 0.0, -0.5069444444444444, -0.3767361111111111, 0.0],
    [-0.3767361111111111, -0.5069444444444444, -0.3767361111111111, -0.
↪5069444444444444, 1.972222222222222, -0.5069444444444444, -0.
↪3767361111111111, -0.5069444444444444, -0.3767361111111111],
    [0.0, -0.3767361111111111, -0.5069444444444444, 0.0, -0.5069444444444444, 1.
↪972222222222222, 0.0, -0.3767361111111111, -0.5069444444444444],
    [0.0, 0.0, 0.0, -0.5069444444444444, -0.3767361111111111, 0.0, 1.
↪972222222222222, -0.5069444444444444, 0.0],
    [0.0, 0.0, 0.0, -0.3767361111111111, -0.5069444444444444, -0.
↪3767361111111111, -0.5069444444444444, 1.972222222222222, -0.
↪5069444444444444],
    [0.0, 0.0, 0.0, 0.0, -0.3767361111111111, -0.5069444444444444, 0.0, -0.
↪5069444444444444, 1.972222222222222],
])
b_4 = np.array([1.4904895819530766, 1.055600747809247, 0.07847904705126368, 0.
↪8311407883427149, 0.0, -0.8765020708205272, -0.6433980946818605, -0.
↪7466392365712349, -0.538021498324083])

# A and b for N=5
A_5 = np.array([
    [2.222222222222222, -0.4444444444444444, 0.0, 0.0, -0.4444444444444444, -0.
↪3611111111111111, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
    [-0.4444444444444444, 2.222222222222222, -0.4444444444444444, 0.0, -0.
↪3611111111111111, -0.4444444444444444, -0.3611111111111111, 0.0, 0.0, 0.0, 0.
↪0, 0.0, 0.0, 0.0, 0.0, 0.0],
    [0.0, -0.4444444444444444, 2.222222222222222, -0.4444444444444444, 0.0, -0.
↪3611111111111111, -0.4444444444444444, -0.3611111111111111, 0.0, 0.0, 0.0, 0.
↪0, 0.0, 0.0, 0.0, 0.0],
    [0.0, 0.0, -0.4444444444444444, 2.222222222222222, 0.0, 0.0, -0.
↪3611111111111111, -0.4444444444444444, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
↪0],
    [-0.4444444444444444, -0.3611111111111111, 0.0, 0.0, 2.222222222222222, -0.
↪4444444444444444, 0.0, 0.0, -0.4444444444444444, -0.3611111111111111, 0.0, 0.
↪0, 0.0, 0.0, 0.0, 0.0],
    [-0.3611111111111111, -0.4444444444444444, -0.3611111111111111, 0.0, -0.
↪4444444444444444, 2.222222222222222, -0.4444444444444444, 0.0, -0.
↪3611111111111111, -0.4444444444444444, -0.3611111111111111, 0.0, 0.0, 0.0, 0.
↪0, 0.0],
    [0.0, -0.3611111111111111, -0.4444444444444444, -0.3611111111111111, 0.0,␣
↪-0.4444444444444444, 2.222222222222222, -0.4444444444444444, 0.0, -0.
↪3611111111111111, -0.4444444444444444, -0.3611111111111111, 0.0, 0.0, 0.0, 0.
↪0],
    [0.0, 0.0, -0.3611111111111111, -0.4444444444444444, 0.0, 0.0, -0.
↪4444444444444444, 2.222222222222222, 0.0, 0.0, -0.3611111111111111, -0.
↪4444444444444444, 0.0, 0.0, 0.0, 0.0],
    [0.0, 0.0, 0.0, 0.0, -0.4444444444444444, -0.3611111111111111, 0.0, 0.0, 2.
↪222222222222222, -0.4444444444444444, 0.0, 0.0, -0.4444444444444444, -0.
↪3611111111111111, 0.0, 0.0],
    [0.0, 0.0, 0.0, 0.0, -0.3611111111111111, -0.4444444444444444, -0.
↪3611111111111111, 0.0, -0.4444444444444444, 2.222222222222222, -0.
↪4444444444444444, 0.0, -0.3611111111111111, -0.4444444444444444, -0.
↪3611111111111111, 0.0],
    [0.0, 0.0, 0.0, 0.0, 0.0, -0.3611111111111111, -0.4444444444444444, -0.
↪3611111111111111, 0.0, -0.4444444444444444, 2.222222222222222, -0.
↪4444444444444444, 0.0, -0.3611111111111111, -0.4444444444444444, -0.
↪3611111111111111],
    [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.3611111111111111, -0.4444444444444444, 0.
↪0, 0.0, -0.4444444444444444, 2.222222222222222, 0.0, 0.0, -0.
↪3611111111111111, -0.4444444444444444],
    [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.4444444444444444, -0.
↪3611111111111111, 0.0, 0.0, 2.222222222222222, -0.4444444444444444, 0.0, 0.
↪0],
    [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.3611111111111111, -0.
↪4444444444444444, -0.3611111111111111, 0.0, -0.4444444444444444, 2.
↪222222222222222, -0.4444444444444444, 0.0],
    [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.3611111111111111, -0.
↪4444444444444444, -0.3611111111111111, 0.0, -0.4444444444444444, 2.
↪222222222222222, -0.4444444444444444],
    [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.3611111111111111, -0.
↪4444444444444444, 0.0, 0.0, -0.4444444444444444, 2.222222222222222],
])
b_5 = np.array([1.2673039440507343, 0.9698054647507671, 1.0133080988552785, 0.
↪07206335813040798, 0.9472174493756345, 0.0, 0.0, -0.9416429716282946, 0.
↪6400834406610956, 0.0, 0.0, -0.7322882523543968, -0.8159823324771336, -0.
↪9192523853093425, -0.48342793699793585, -0.19471066818706848])
```

First I'll implement the function $g(x, y)$:

$$g(x,y) = \begin{cases} \sin(4y) & \text{if } x = 0, \\ \sin(3x) & \text{if } y = 0, \\ \sin(3+4y) & \text{if } x = 1, \\ \sin(3x+4) & \text{if } y = 1. \end{cases}$$

[3]:
```python
def g(x, y):

    return np.sin(3*x + 4*y)
```

The next function is to help with converting a point's coordinate on the grid into a row/column index for a matrix:

[4]:
```python
def pj_to_col(pj, N):

    #Remove any points which out out of bounds of the interior grid
    pj = pj[(pj[:,0] >= 0) & (pj[:,1] >= 0)
                                    & (pj[:,0] < (N-1)) & (pj[:,1] < (N-1))]

    #Convert the grid indices into row/column indices
    return pj[:,0]*(N-1) + pj[:,1]
```

[5]:
```python
def helmholtz_fe(N):



    ###############################################
    #Set up constant values
    h = 1/N
    k = 5

    #Values for different entries
    diagonal = (24-4*h**2*k**2)/9
    hv_adjacent = (-3-1*h**2*k**2)/9
    d_adjacent = (-12-1*h**2*k**2)/36

    b_term1 = (12+h**2*k**2)/36
    b_term2 = (3+h**2*k**2)/9

    ###############################################
    #Initialize data storage

    #Number of elements is:
    #(N-1)^2 for the diagonal, 1 diagonal
    #The corners have 3 neighbors (1 diagonal, 1 horizontal, 1 vertical), there␣
 ↪are 4 corners
    #edge points have 5 neighbors (2 diagonal, 3 mix of horizontal and␣
 ↪vertical), there are 4(N-3) edge points
    #all other points have 8 neighbors (4 diagonal, 2 horizontal, 2 vertical),␣
 ↪there are (N-1)^2 - 4(N-2) other points
    nelements = (N-1)**2 + 4*3 + (4*(N-3))*5 + ((N-1)**2 - 4*(N-2))*8

    #Set up coo styled data storage
    row_ind = np.zeros(nelements, dtype=int)
    col_ind = np.zeros(nelements, dtype=int)
    data = np.zeros(nelements, dtype=np.float64)

    #Create the empty vector
    b = np.zeros((N-1)**2, dtype=np.float64)
```

```
␣
↪############################################################################################
    #Iterate through the matrix rows
    count = 0
    for row in range(0,(N-1)**2):

        #Work out p_(row) (starts at (0,0) meaning (x=h, y=h) then -> (0,1)␣
↪meaning (x=2h, y=h) )
        p_i = (row//(N-1), row%(N-1))

        ################################################
        #Find columns where p_(row) is horizontally or vertically adjacent to␣
↪p_(column)
        adjs = np.array([[p_i[0],p_i[1]-1],
                        [p_i[0],p_i[1]+1],
                        [p_i[0]-1,p_i[1]],
                        [p_i[0]+1,p_i[1]]])

        adjacent_cols = pj_to_col(adjs,N)

        n_adjacent = len(adjacent_cols)
        col_ind[count+1:count+1+n_adjacent] = adjacent_cols
        data[count+1:count+1+n_adjacent] = hv_adjacent

        ################################################
        #Find columns where p_(row) is diagonally adjacent to p_(column)
        diags = np.array([[p_i[0]+1,p_i[1]+1],
                         [p_i[0]-1,p_i[1]-1],
                         [p_i[0]+1,p_i[1]-1],
                         [p_i[0]-1,p_i[1]+1]])

        diagonal_cols = pj_to_col(diags,N)

        n_diagonal = len(diagonal_cols)
        col_ind[count+1+n_adjacent:count+1+n_adjacent+n_diagonal] =␣
↪diagonal_cols
        data[count+1+n_adjacent:count+1+n_adjacent+n_diagonal] = d_adjacent

        ##################################################
        #Save the diagonals

        #Work out total number of non zero elements in the row
        n_in_row = 1+n_adjacent+n_diagonal

        #Save all row indices
        row_ind[count : count+n_in_row] = row
        #Save the diagonal
        col_ind[count] = row
        data[count] = diagonal

        count += n_in_row

        ###############################################
        #Save the vector data

        #Rescale the adjacent and diagonal values so that they account for h
        adjs = adjs*h+h
        diags = diags*h+h

        #Only keep adjacent and diagonal points which are on the boundary (have␣
↪one coordinate a 0 or 1)
        adjs = adjs[(adjs[:,0] == 0) | (adjs[:,0] == 1) | (adjs[:,1] == 0) |␣
↪(adjs[:,1] == 1)]
        diags = diags[(diags[:,0] == 0) | (diags[:,0] == 1) | (diags[:,1] == 0)␣
↪| (diags[:,1] == 1) ]

        #Compute b_j
        b[row] = b_term1*np.sum(g(diags[:,1], diags[:,0])) + b_term2*np.
↪sum(g(adjs[:,1], adjs[:,0]))

        ###############################################

  ␣
↪###########################################################################################


    return coo_matrix((data, (row_ind, col_ind)), shape=((N-1)**2, (N-1)**2)).
↪tocsr(), b
```

A quick check to see if the matrix and vector are as expected:

[6]:
```
assert np.allclose(helmholtz_fe(2)[0].toarray(), A_2)
assert np.allclose(helmholtz_fe(3)[0].toarray(), A_3)
assert np.allclose(helmholtz_fe(4)[0].toarray(), A_4)
assert np.allclose(helmholtz_fe(5)[0].toarray(), A_5)
```

[7]:
```
assert np.allclose(helmholtz_fe(2)[1], b_2)
assert np.allclose(helmholtz_fe(3)[1], b_3)
assert np.allclose(helmholtz_fe(4)[1], b_4)
assert np.allclose(helmholtz_fe(5)[1], b_5)
```

1.2 Part 2: solving the system

Solving the matrix-vector problem will lead to an approximate solution to the Helmholtz problem: we call this approximate solution u_h.

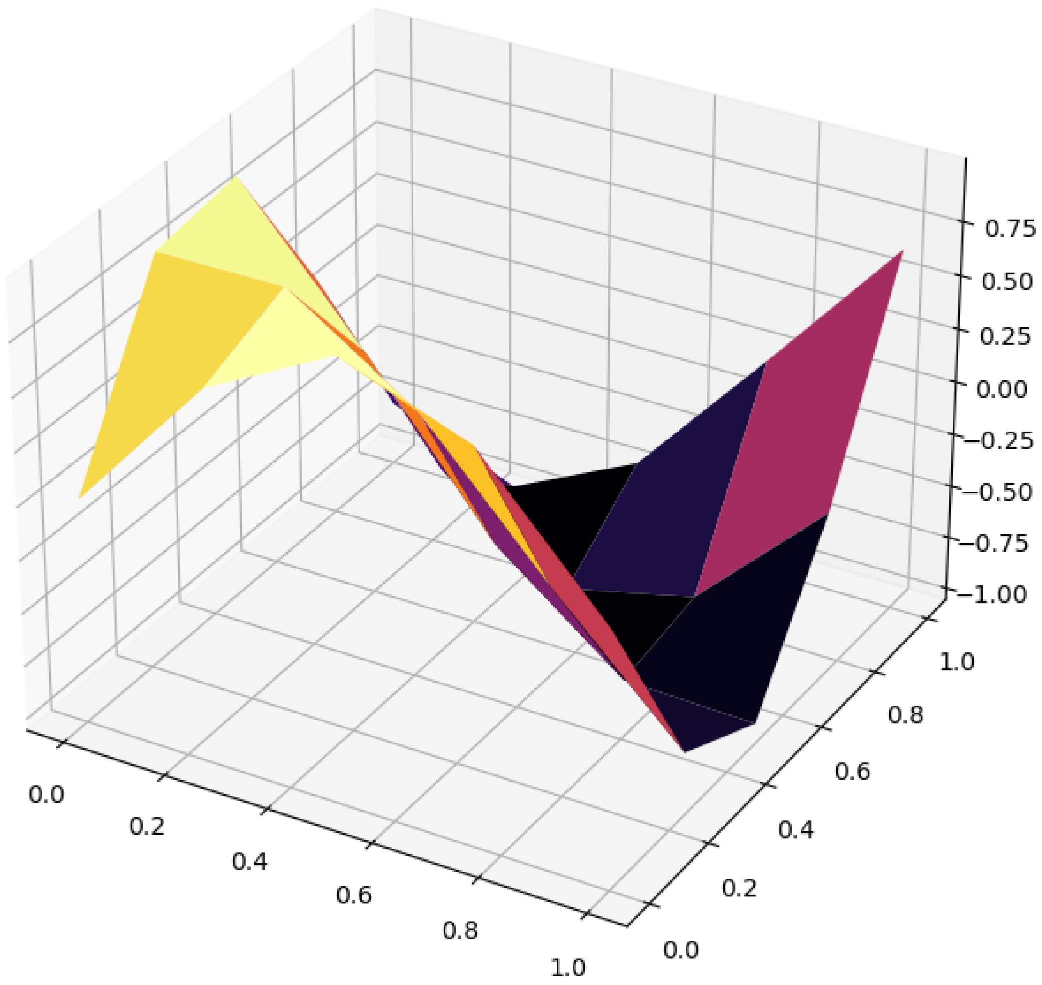
Using any matrix-vector solver, **solve the matrix-vector problem for $N = 4$, $N = 8$, and $N = 16$** and **plot the approximate solutions to the Helmholtz problem**. To plot the solutions, you can pass the x- and y-coordinates of the points and the value of u_h at each point into matplotlib's 3D plotting function. For the points on the boundary, the value of u_h is given by the function g; for interior points, the value will be one of the entries of the solution vector $\mathbf{x}$.

An example of 3D plotting in matplotlib can be found in the sparse PDE example from earlier in the course.

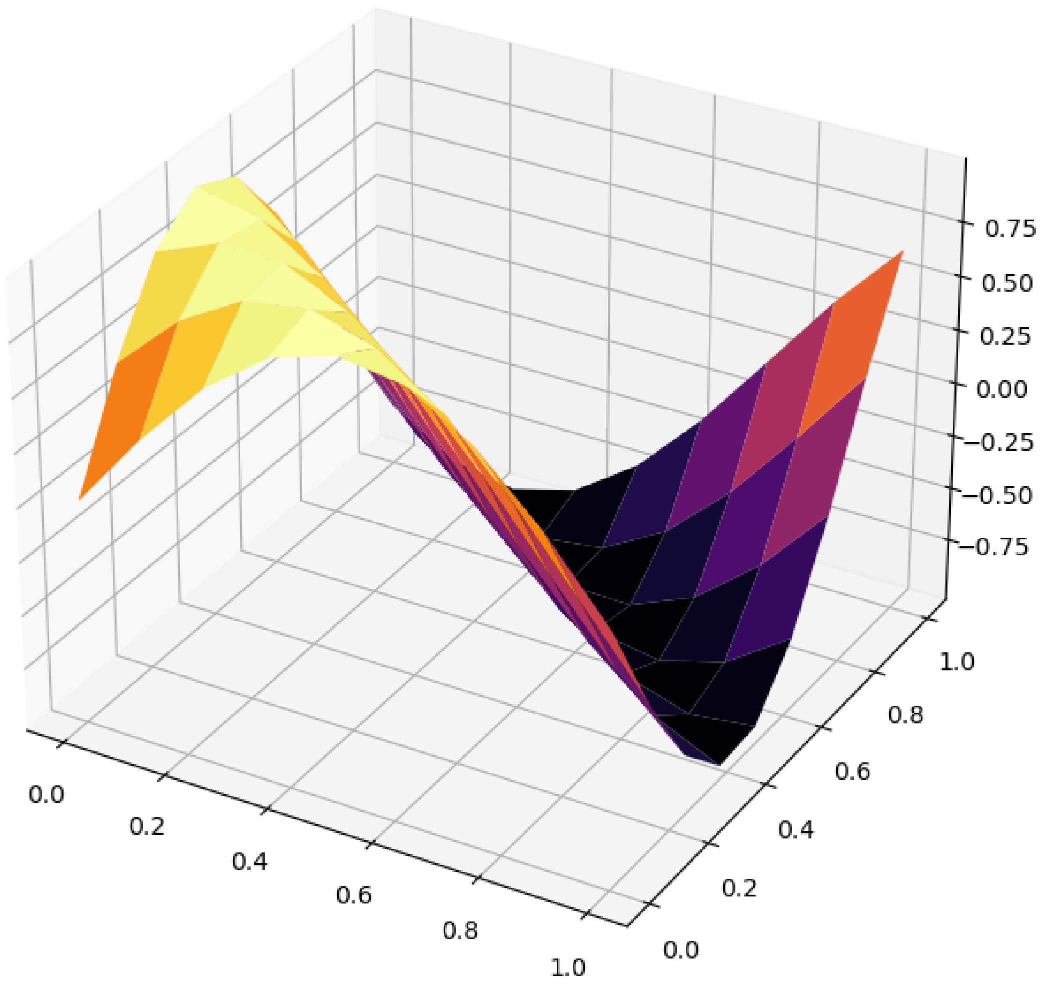
[8]:
```
from scipy.sparse.linalg import spsolve
```

[9]:
```
def plot_solution(N):

    #Generate the matrix and vector
    A, f = helmholtz_fe(N)

    #Solve the problem for the interior points - using scipy spsolve
    sol = spsolve(A, f)
    u = sol.reshape((N-1, N-1))

    #Evaluate the boundary points
    ticks= np.linspace(0, 1, N+1)
    X, Y = np.meshgrid(ticks, ticks)
    u_full = g(X, Y) #This evaluates g for the full grid
    u_full[1:N, 1:N] = u #Replace the interior points with the matrix vector␣
 ↪solutions

    #Plot
    fig = plt.figure(figsize=(8, 8),dpi=100)
    ax = plt.axes(projection='3d')
    surf = ax.plot_surface(X, Y, u_full, antialiased=True, cmap=cm.inferno)
    plt.show()
```

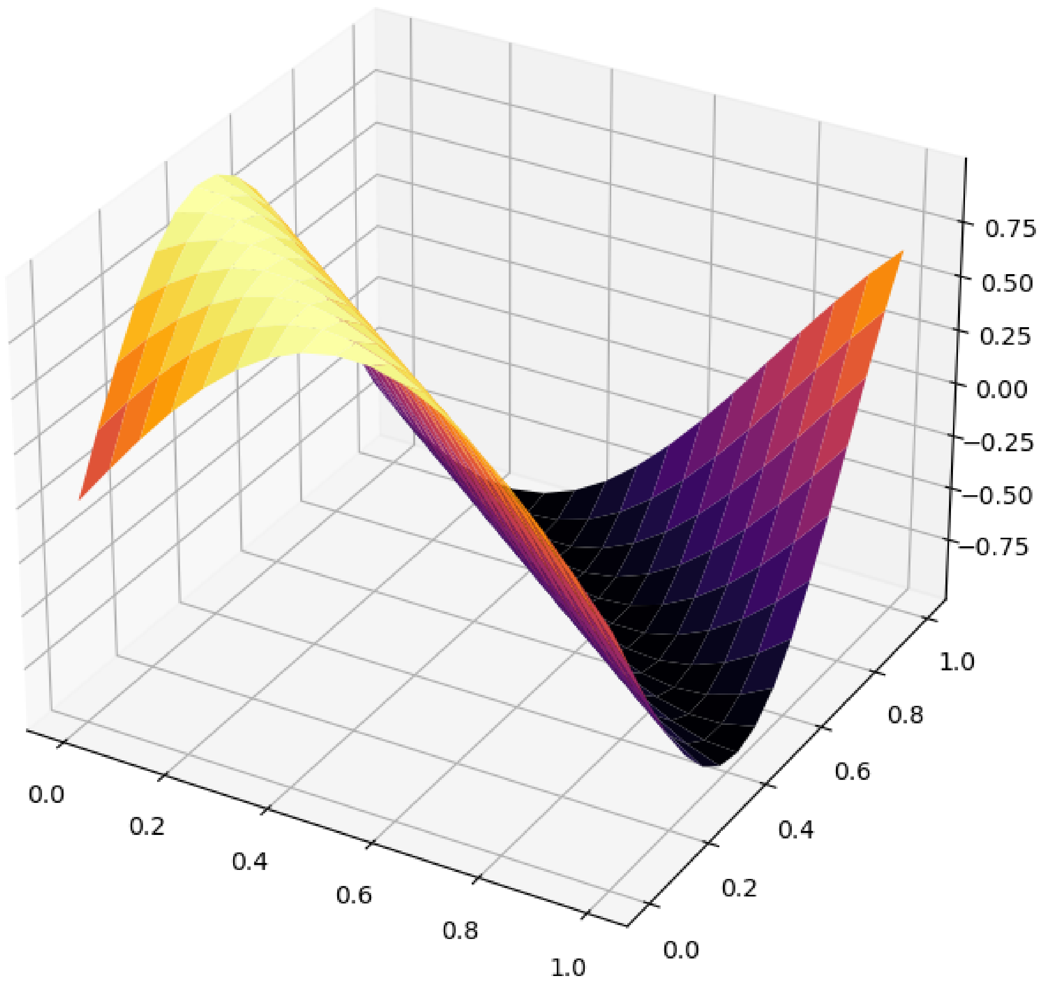
[10]:
```
plot_solution(4)
```



[11]:
```
plot_solution(8)
```



[12]:
```
plot_solution(16)
```



1.3 Part 3: comparing solvers and preconditioners

In this section, your task is to evaluate the performance of various matrix-vector solvers. To do this, **solve the matrix-vector problem with small to medium sized value of N using a range of different solvers of your choice, measuring factors you deem to be important for your evaluation.** These factors should include the time taken by the solver, and may additionally include many other thing such as the number of iterations taken by an iterative solver, or the size of the residual after each iteration. **Make a set of plots that show the measurements you have made and allow you to compare the solvers**.

You should compare at least five matrix-vector solvers: at least two of these should be iterative solvers, and at least one should be a direct solver. You can use solvers from the Scipy library. (You may optionally use additional solvers from other linear algebra libraries such as PETSc, but you do not need to do this to achieve high marks. You should use solvers from these libraries and do not need to implement your own solvers.) For two of the iterative solvers you have chosen to use, **repeat the comparisons with three different choices of preconditioner**.

[13]:
```
import pandas as pd
from petsc4py import PETSc
```

For this section I will use the following matrix vector problem solvers:

Solver	type
`scipy.gmres`	iterative
`scipy.cg`	iterative
`scipy.bicgstab`	iterative
`scipy.spsolve`	direct
`scipy.splu`	direct
`PETSc` + preconditioning	iterative with preconditioning
`PETSc` + preconditioning	iterative with preconditioning

I will evaluate them on the following set of metrics: `{run time, number of iterations required, final residual, residual per iteration for the largest matrix}` Below is an object for keeping track of the number of iterations an iterative solver took and the residual at each iteration:

[14]:
```
class counter(object):
    def __init__(self):
        self.niter = 0
        self.residuals = []
    def __call__(self, rk):
        self.niter += 1
        self.residuals.append(rk)
```

[15]:
```
#We will use a range of ns from 2 up to 49
Ns = np.arange(2,50)
```

1.3.1 Testing the Scipy Solvers

First we will look at some iterative solvers

gmres

[16]:
```
from scipy.sparse.linalg import gmres
```

[17]:
```
gmres_times = []
gmres_iters = []
gmres_residuals = []

for N in Ns:

    #Create the matrix
    A, f = helmholtz_fe(N)
    count = counter()
    #Check run time
    start = perf_counter()
    sol = gmres(A, f, callback=count)
    gmres_times.append(perf_counter() - start)

    #Count the number of iterations it took to run
    gmres_iters.append(count.niter)

    #Save the final residual
    gmres_residuals.append(sol[1])

gmres_residual_by_iter = count.residuals
gmres_sol = sol[0]
```

cg

[18]:
```
from scipy.sparse.linalg import cg
```

[19]:
```
cg_times = []
cg_iters = []
cg_residuals = []

for N in Ns:

    A, f = helmholtz_fe(N)
    count = counter()

    #Check run time
    start = perf_counter()
    sol = cg(A, f, callback=count)
    cg_times.append(perf_counter() - start)

    #Count the number of iterations it took to run
    cg_iters.append(count.niter)

    #Save the final residual
    cg_residuals.append(sol[1])

cg_residual_by_iter = count.residuals[0]
cg_sol = sol[0]
```

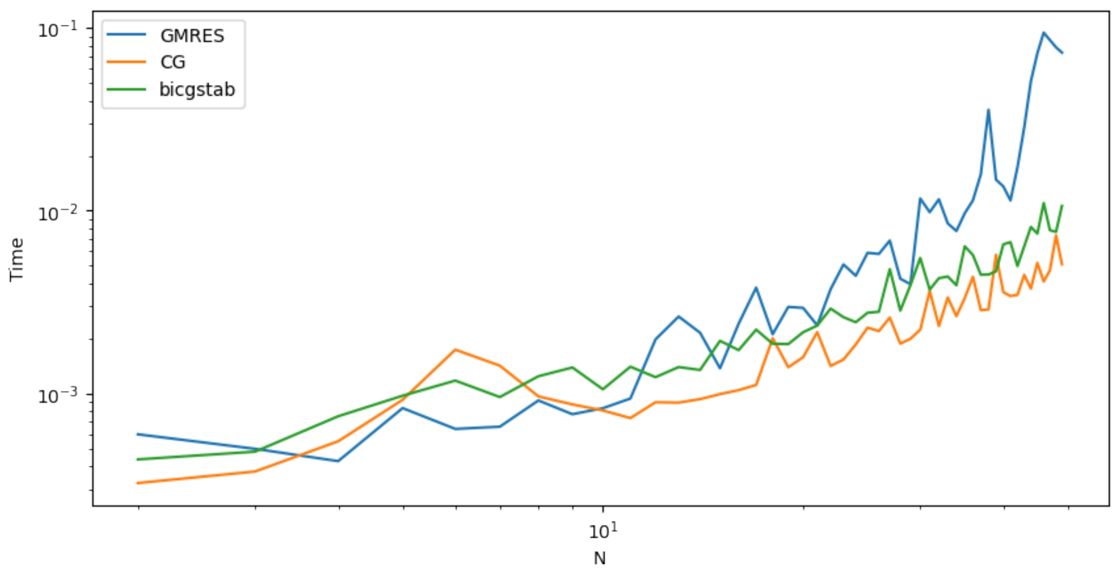
bicgstab

[20]:
```
from scipy.sparse.linalg import bicgstab
```

[21]:
```
bicgstab_times = []
bicgstab_iters = []
bicgstab_residuals = []

for N in Ns:

    A, f = helmholtz_fe(N)
    count = counter()

    #Check run time
    start = perf_counter()
    sol = bicgstab(A, f, callback=count)
    bicgstab_times.append(perf_counter() - start)

    #Count the number of iterations it took to run
    bicgstab_iters.append(count.niter)

    #Save the final residual
    bicgstab_residuals.append(sol[1])

bicgstab_residual_by_iter = count.residuals[0]
bicgstab_sol = sol[0]
```

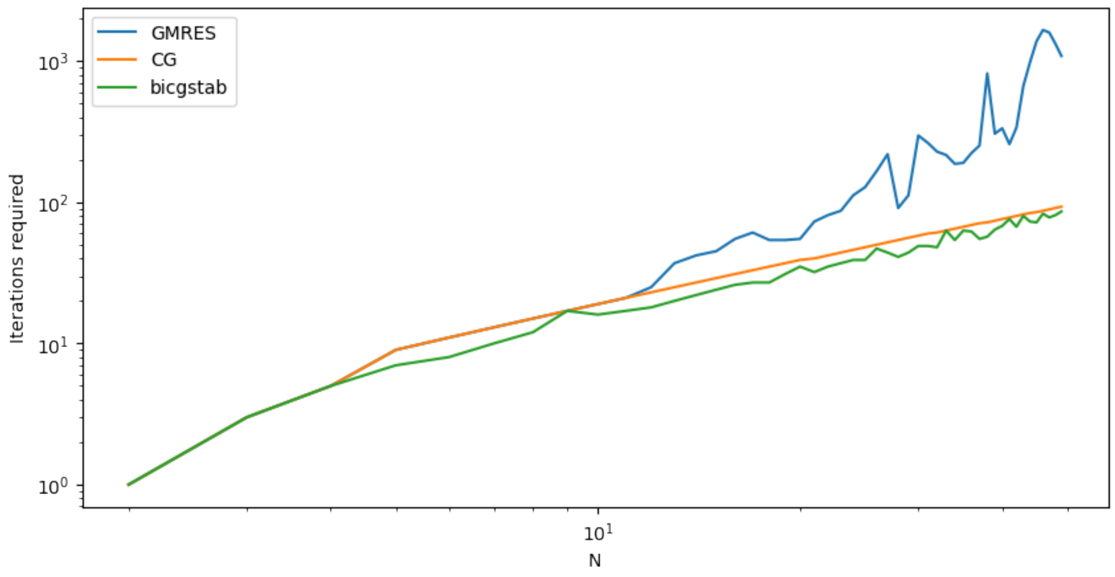
We can compare the iterative scipy solvers to see which one is the fastest

[22]:
```
fig, ax = plt.subplots(figsize=(10, 5),dpi=100)
ax.plot(Ns, gmres_times, label = "GMRES")
ax.plot(Ns, cg_times, label = "CG")
ax.plot(Ns, bicgstab_times, label = "bicgstab")
ax.set_xlabel("N")
ax.set_ylabel("Time")
ax.set_xscale("log")
ax.set_yscale("log")
ax.legend();
```



It looks like GMRES is performing the slowest while cg is the quickest of the iterative scipy solvers. If we take a look at the number of iterations required to arrive at a solution compared to the matrix size we can see that this grows much faster for GMRES compared to cg and bicgstab:

[23]:
```
fig, ax = plt.subplots(figsize=(10, 5),dpi=100)
ax.plot(Ns, gmres_iters, label = "GMRES")
ax.plot(Ns, cg_iters, label = "CG")
ax.plot(Ns, bicgstab_iters, label = "bicgstab")

ax.set_xlabel("N")
ax.set_ylabel("Iterations required")
ax.set_xscale("log")
ax.set_yscale("log")
ax.legend();
```



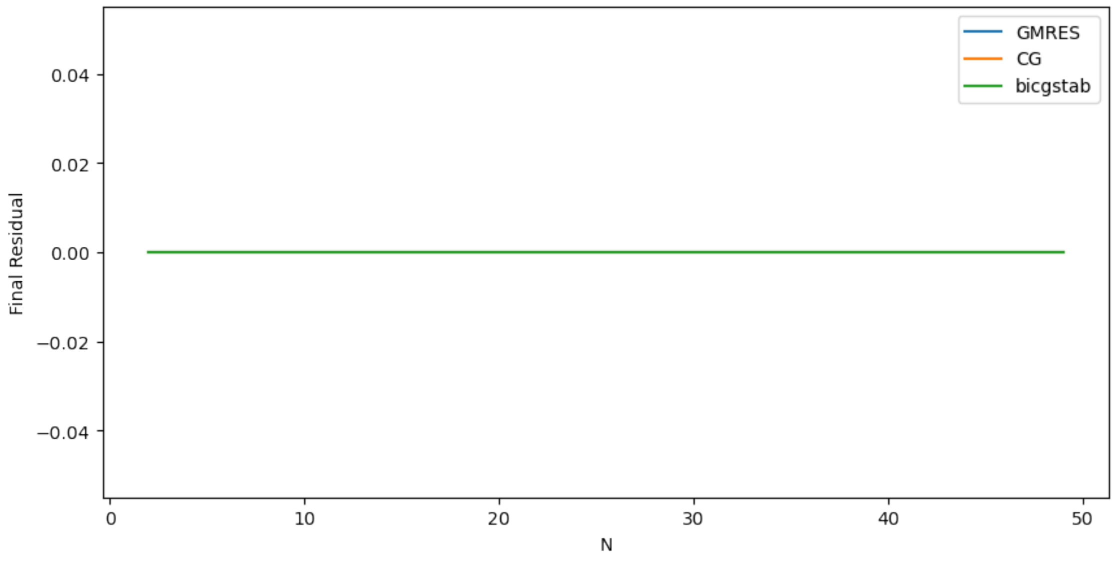
Both cg and bicgstab are conjugate gradient methods and rely on the assumption that the matrix is positive definite. If this is not the case, the solvers may struggle to converge. We can check to see if the values of the solutions agree:

[24]:
```
print(f"Difference between gmres and cg solution: {np.
  ↪mean(abs(gmres_sol-cg_sol))}")
print(f"Difference between gmres and bicgstab solution: {np.
  ↪mean(abs(gmres_sol-bicgstab_sol))}")
print(f"Difference between bicgstab and cg solution: {np.
  ↪mean(abs(bicgstab_sol-cg_sol))}")
```

```
Difference between gmres and cg solution: 0.0006392283445148072
Difference between gmres and bicgstab solution: 0.0006854768249450281
Difference between bicgstab and cg solution: 5.009813752391096e-05
```

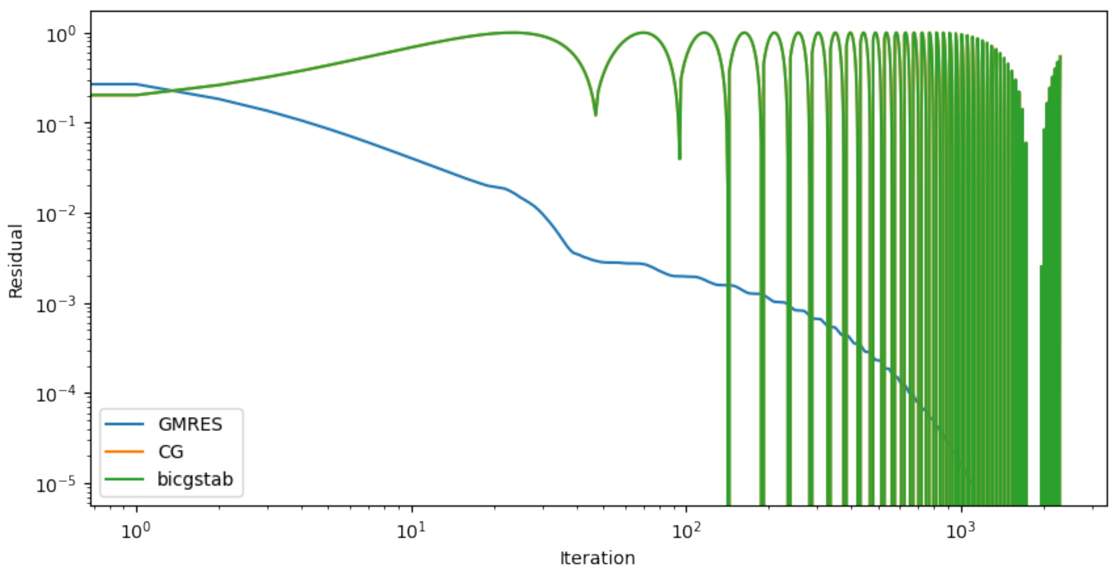
It looks like bicgstab and cg agree with each other more than they both agree with gmres. Lets check the final residuals of each method to see if they all converged:

[25]:
```
fig, ax = plt.subplots(figsize=(10, 5),dpi=100)
ax.plot(Ns, gmres_residuals, label = "GMRES")
ax.plot(Ns, cg_residuals, label = "CG")
ax.plot(Ns, bicgstab_residuals, label = "bicgstab")

ax.set_xlabel("N")
ax.set_ylabel("Final Residual")
ax.legend();
```



It looks like all of the solvers are converging. However if we look at the residual per iteration of the largest matrix we can see that the ways in which they converge differ:

[26]:
```
fig, ax = plt.subplots(figsize=(10, 5),dpi=100)
ax.plot(gmres_residual_by_iter, label = "GMRES")
ax.plot(cg_residual_by_iter, label = "CG")
ax.plot(bicgstab_residual_by_iter, label = "bicgstab")

ax.set_xlabel("Iteration")
ax.set_ylabel("Residual")
ax.set_xscale("log")
ax.set_yscale("log")
ax.legend();
```



The green line covers the result for cg as well - it appears that cg and bicgstab are oscillating around the solution while gmres converges in more stable way. This is likely due to gradient descent like method cg uses - it gets very close to the exact solution and then takes a step perpendicular to the vector pointing in the direction where the gradient is steepest - this sends the solver away from the solution and increases the residual.

We will now take a look at some direct scipy solvers:

Testing scipy.spsolve

[27]:
```
spsolve_times = []

for N in Ns:

    A, f = helmholtz_fe(N)

    #Check run time
    start = perf_counter()
    sol = spsolve(A, f)
    spsolve_times.append(perf_counter() - start)

spsolve_sol = sol
```

Testing scipy.splu.solve()

[28]:
```
from scipy.sparse.linalg import splu
```

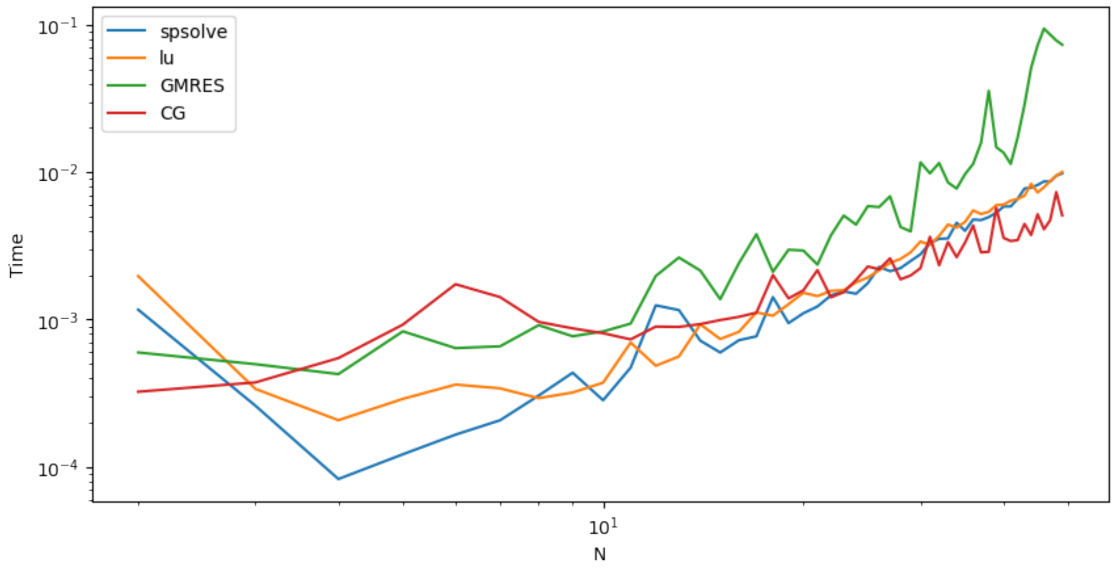
[29]:
```
splu_times = []
```

```
for N in Ns:

    A, f = helmholtz_fe(N)

    #Check run time
    start = perf_counter()
    sol = splu(A).solve(f)
    splu_times.append(perf_counter() - start)

lu_sol = sol
```

```
/Users/james/opt/anaconda3/lib/python3.8/site-
packages/scipy/sparse/linalg/_dsolve/linsolve.py:347: SparseEfficiencyWarning:
splu converted its input to CSC format
  warn('splu converted its input to CSC format', SparseEfficiencyWarning)
```

Lets see how fast these solvers run and compare them to gmres and cg

[30]:
```
fig, ax = plt.subplots(figsize=(10, 5),dpi=100)
ax.plot(Ns, spsolve_times, label = "spsolve")
ax.plot(Ns, splu_times, label = "lu")
ax.plot(Ns, gmres_times, label = "GMRES")
ax.plot(Ns, cg_times, label = "CG")
ax.set_xlabel("N")
ax.set_ylabel("Time")
ax.set_xscale("log")
ax.set_yscale("log")
ax.legend();
```



It appears that the direct solvers are faster than gmres but as the matrix sizes become larger they become slower than cg. Let us see how their solutions differ to the iterative solvers:

[31]:
```
print(f"Difference between spsolve and lu solution: {np.
  ↪mean(abs(spsolve_sol-lu_sol))}")
print(f"Difference between spsolve and gmres solution: {np.
  ↪mean(abs(gmres_sol-spsolve_sol))}")
print(f"Difference between spsolve and bicgstab solution: {np.
  ↪mean(abs(spsolve_sol-bicgstab_sol))}")
print(f"Difference between spsolve and cg solution: {np.
  ↪mean(abs(spsolve_sol-cg_sol))}")
```

```
Difference between spsolve and lu solution: 2.3205926368996276e-16
Difference between spsolve and gmres solution: 0.000639616744547426
Difference between spsolve and bicgstab solution: 4.9283881289306196e-05
Difference between spsolve and cg solution: 4.319592881767775e-06
```

As expected, the direct solvers agree with each other up to the precision of double floating point numbers. Interestingly, the direct solvers agree closest with the solutions produced by cg and disagree the most with gmres. This means that although the convergence of cg appears to be much less smooth than gmres, its solutions are closer to the exact solutions.

1.3.2 Testing with PETSc solvers

We will now take a look at some solvers from the PETSc library, for each of these solvers we will also use three different preconditioners:

PETSc.bcgs

[32]:
```
bcgs1_times = []
bcgs1_iters = []
bcgs1_residuals = []

for N in Ns:

    #Setup our solver
    ksp = PETSc.KSP().create()
    ksp.setType("bcgs")

    A, f = helmholtz_fe(N)
    #Convert matrix and vector to PETSc
    A = PETSc.Mat().createAIJ(size=A.shape,csr=(A.indptr, A.indices,A.data))
    b = A.createVecLeft()
    b.array[:] = f

    #Vector to solve
    x = A.createVecRight()

    ksp.setOperators(A)
    ksp.setConvergenceHistory()
    ksp.getPC().setType("none") #No preconditioning

    #Solve the system
    start = perf_counter()
    ksp.solve(b, x)
    bcgs1_times.append(perf_counter() - start)

    #Get convergence info
    res = ksp.getConvergenceHistory()
    bcgs1_iters.append(len(res))
    bcgs1_residuals.append(res[-1])
```

[33]:
```
bcgs2_times = []
bcgs2_iters = []
bcgs2_residuals = []

for N in Ns:

    #Setup our solver
    ksp = PETSc.KSP().create()
    ksp.setType("bcgs")

    A, f = helmholtz_fe(N)

    #Convert matrix and vector to PETSc
    A = PETSc.Mat().createAIJ(size=A.shape,csr=(A.indptr, A.indices,A.data))
    b = A.createVecLeft()
    b.array[:] = f

    #Vector to solve
    x = A.createVecRight()

    ksp.setOperators(A)
    ksp.setConvergenceHistory()
    ksp.getPC().setType("ilu")

    #Solve the system
    start = perf_counter()
    ksp.solve(b, x)
    bcgs2_times.append(perf_counter() - start)

    #Get convergence info
    res = ksp.getConvergenceHistory()
    bcgs2_iters.append(len(res))
    bcgs2_residuals.append(res[-1])
```

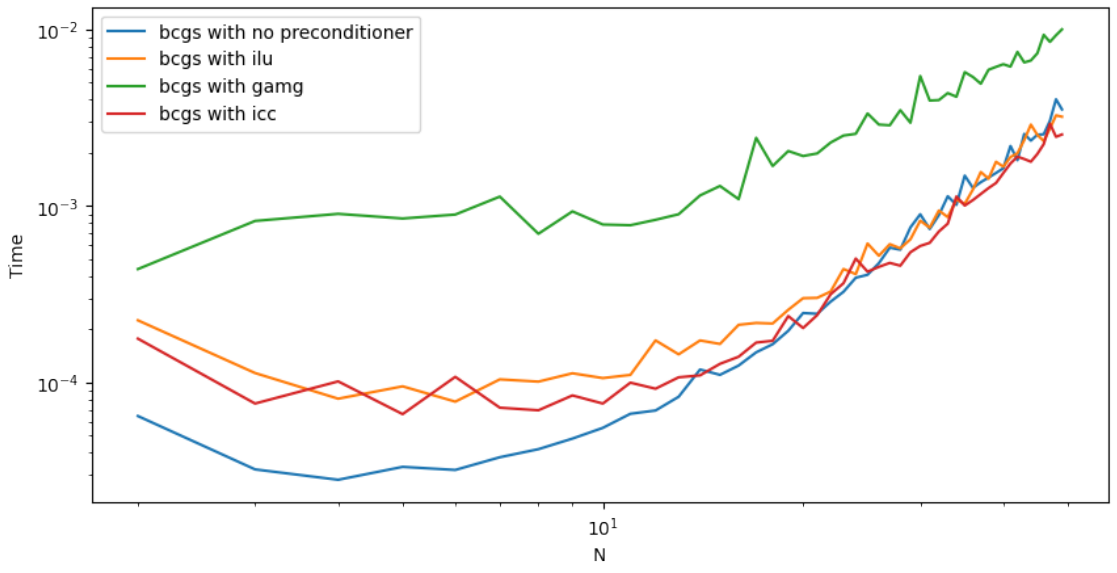
```
bcgs_ilu_sol = ksp.getSolution().getArray()
```

[34]:
```
bcgs3_times = []
bcgs3_iters = []
bcgs3_residuals = []

for N in Ns:

    #Setup our solver
    ksp = PETSc.KSP().create()
    ksp.setType("bcgs")

    A, f = helmholtz_fe(N)

    #Convert matrix and vector to PETSc
    A = PETSc.Mat().createAIJ(size=A.shape,csr=(A.indptr, A.indices,A.data))
    b = A.createVecLeft()
    b.array[:] = f

    #Vector to solve
    x = A.createVecRight()

    ksp.setOperators(A)
    ksp.setConvergenceHistory()
    ksp.getPC().setType("gamg")

    #Solve the system
    start = perf_counter()
    ksp.solve(b, x)
    bcgs3_times.append(perf_counter() - start)

    #Get convergence info
    res = ksp.getConvergenceHistory()
    bcgs3_iters.append(len(res))
    bcgs3_residuals.append(res[-1])

bcgs_gamg_sol = ksp.getSolution().getArray()
```

[35]:
```
bcgs4_times = []
bcgs4_iters = []
bcgs4_residuals = []

for N in Ns:

    #Setup our solver
    ksp = PETSc.KSP().create()
    ksp.setType("bcgs")
    A, f = helmholtz_fe(N)

    #Convert matrix and vector to PETSc
    A = PETSc.Mat().createAIJ(size=A.shape,csr=(A.indptr, A.indices,A.data))
    b = A.createVecLeft()
    b.array[:] = f

    #Vector to solve
    x = A.createVecRight()

    ksp.setOperators(A)
    ksp.setConvergenceHistory()
    ksp.getPC().setType("icc")


    #Solve the system
    start = perf_counter()
    ksp.solve(b, x)
    bcgs4_times.append(perf_counter() - start)

    #Get convergence info
    res = ksp.getConvergenceHistory()
    bcgs4_iters.append(len(res))
    bcgs4_residuals.append(res[-1])
```

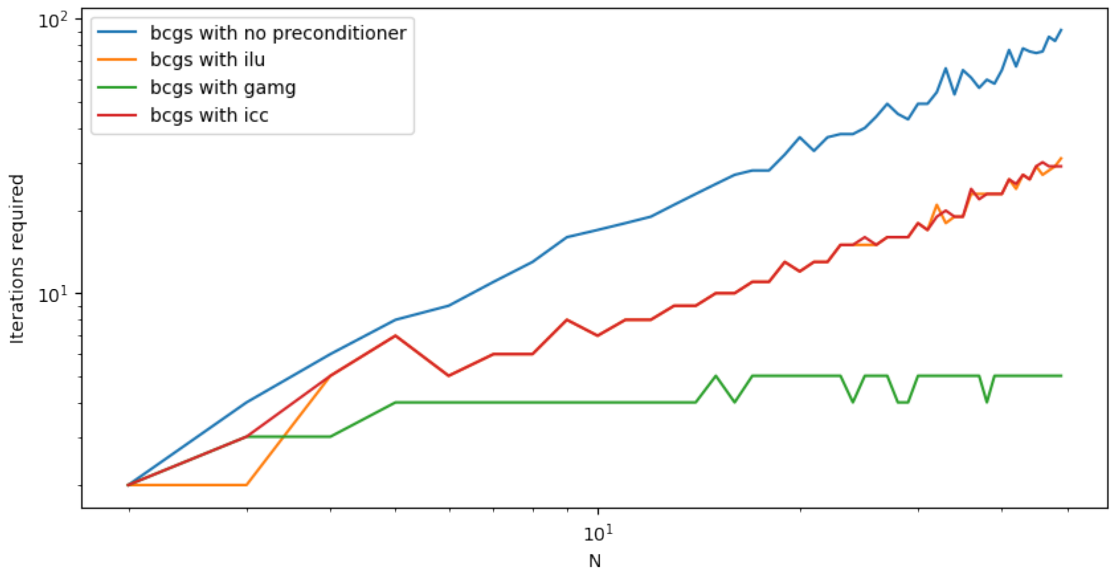
Let us take a look at how this solver performed with different preconditioners:

[36]:
```
fig, ax = plt.subplots(figsize=(10, 5),dpi=100)
ax.plot(Ns, bcgs1_times, label = "bcgs with no preconditioner")
ax.plot(Ns, bcgs2_times, label = "bcgs with ilu")
ax.plot(Ns, bcgs3_times, label = "bcgs with gamg")
ax.plot(Ns, bcgs4_times, label = "bcgs with icc")
ax.set_xlabel("N")
ax.set_ylabel("Time")
ax.set_xscale("log")
ax.set_yscale("log")
ax.legend();
```



It looks like as the matrix size grows, the preconditioned solvers begin to out perform the un-preconditioned solver. icc performs the best, followed by incomplete lu. gamag performs far worse for this solver for this range of matrix sizes than any of the other preconditioners.

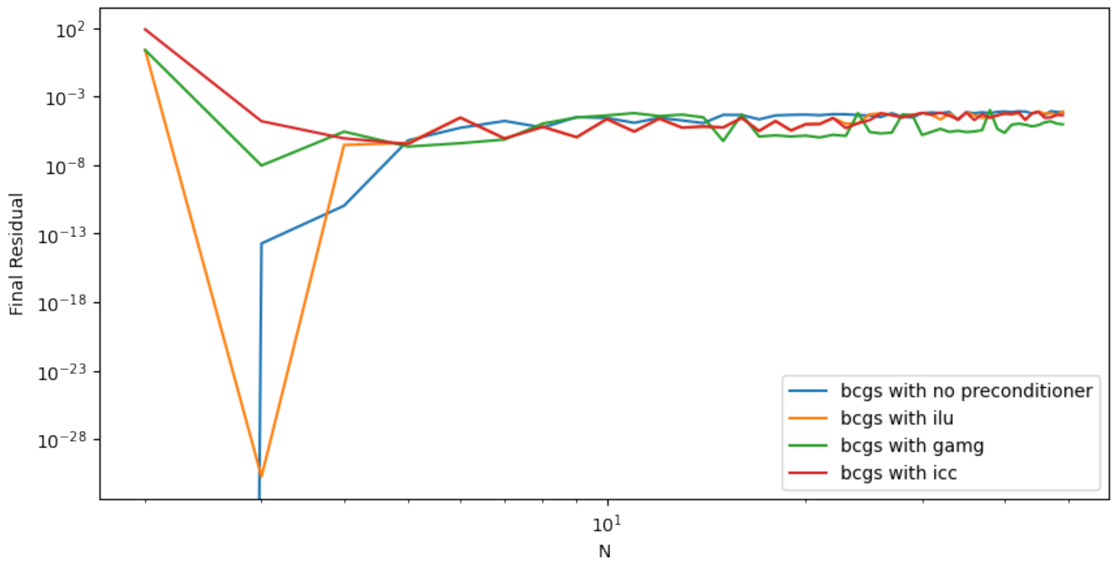
[37]:
```
fig, ax = plt.subplots(figsize=(10, 5),dpi=100)
ax.plot(Ns, bcgs1_iters, label = "bcgs with no preconditioner")
ax.plot(Ns, bcgs2_iters, label = "bcgs with ilu")
ax.plot(Ns, bcgs3_iters, label = "bcgs with gamg")
ax.plot(Ns, bcgs4_iters, label = "bcgs with icc")
ax.set_xlabel("N")
ax.set_ylabel("Iterations required")
ax.set_xscale("log")
ax.set_yscale("log")
ax.legend();
```



It looks like the number of iterations required for gamg grows at a far slower rate than the other methods - this is most likely because the preconditioner does most of the work allowing the solver the perform relatively few iterations.

Let us quickly check that all of the methods converged:

[38]:
```
fig, ax = plt.subplots(figsize=(10, 5),dpi=100)
ax.plot(Ns, bcgs1_residuals, label = "bcgs with no preconditioner")
ax.plot(Ns, bcgs2_residuals, label = "bcgs with ilu")
ax.plot(Ns, bcgs3_residuals, label = "bcgs with gamg")
ax.plot(Ns, bcgs4_residuals, label = "bcgs with icc")
ax.set_xlabel("N")
ax.set_ylabel("Final Residual")
ax.set_xscale("log")
ax.set_yscale("log")
ax.legend();
```



It looks like all of the methods converge, with the final residual being very small. We will now look at a second PETSc solver:

Testing PETSc.chebyshev

[39]:
```
chebyshev1_times = []
chebyshev1_iters = []
chebyshev1_residuals = []

for N in Ns:

    #Setup our solver
    ksp = PETSc.KSP().create()
    ksp.setType("chebyshev")

    A, f = helmholtz_fe(N)

    #Convert matrix and vector to PETSc
    A = PETSc.Mat().createAIJ(size=A.shape,csr=(A.indptr, A.indices,A.data))
    b = A.createVecLeft()
    b.array[:] = f

    #Vector to solve
    x = A.createVecRight()

    ksp.setOperators(A)
    ksp.setConvergenceHistory()
    ksp.getPC().setType("none") #No preconditioning
    start = perf_counter()

    #Solve the system
    ksp.solve(b, x)
    chebyshev1_times.append(perf_counter() - start)

    #Get convergence info
    res = ksp.getConvergenceHistory()
    chebyshev1_iters.append(len(res))
    chebyshev1_residuals.append(res[-1])
```

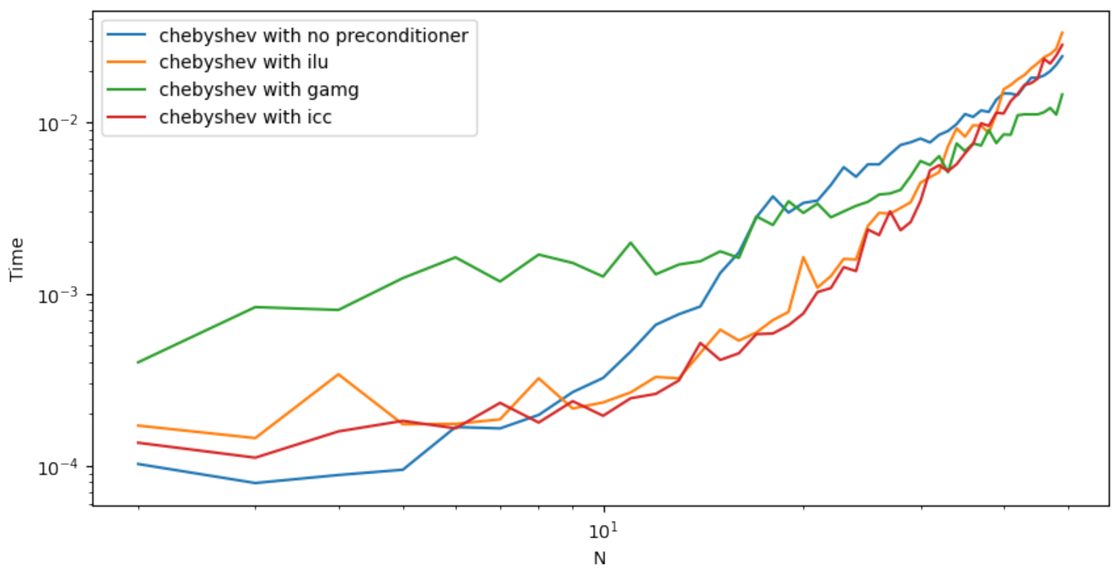
[40]:

```
chebyshev2_times = []
chebyshev2_iters = []
chebyshev2_residuals = []

for N in Ns:

    #Setup our solver
    ksp = PETSc.KSP().create()
    ksp.setType("chebyshev")

    A, f = helmholtz_fe(N)

    #Convert matrix and vector to PETSc
    A = PETSc.Mat().createAIJ(size=A.shape,csr=(A.indptr, A.indices,A.data))
    b = A.createVecLeft()
    b.array[:] = f

    #Vector to solve
    x = A.createVecRight()

    ksp.setOperators(A)
    ksp.setConvergenceHistory()
    ksp.getPC().setType("ilu")

    start = perf_counter()

    #Solve the system
    ksp.solve(b, x)
    chebyshev2_times.append(perf_counter() - start)

    #Get convergence info
    res = ksp.getConvergenceHistory()
    chebyshev2_iters.append(len(res))
    chebyshev2_residuals.append(res[-1])
```

[41]:
```
chebyshev3_times = []
chebyshev3_iters = []
chebyshev3_residuals = []

for N in Ns:

    #Setup our solver
    ksp = PETSc.KSP().create()
    ksp.setType("chebyshev")

    A, f = helmholtz_fe(N)

    #Convert matrix and vector to PETSc
    A = PETSc.Mat().createAIJ(size=A.shape,csr=(A.indptr, A.indices,A.data))
    b = A.createVecLeft()
    b.array[:] = f

    #Vector to solve
    x = A.createVecRight()

    ksp.setOperators(A)
    ksp.setConvergenceHistory()
    ksp.getPC().setType("gamg")

    start = perf_counter()

    #Solve the system
    ksp.solve(b, x)
    chebyshev3_times.append(perf_counter() - start)

    #Get convergence info
    res = ksp.getConvergenceHistory()
    chebyshev3_iters.append(len(res))
    chebyshev3_residuals.append(res[-1])
```

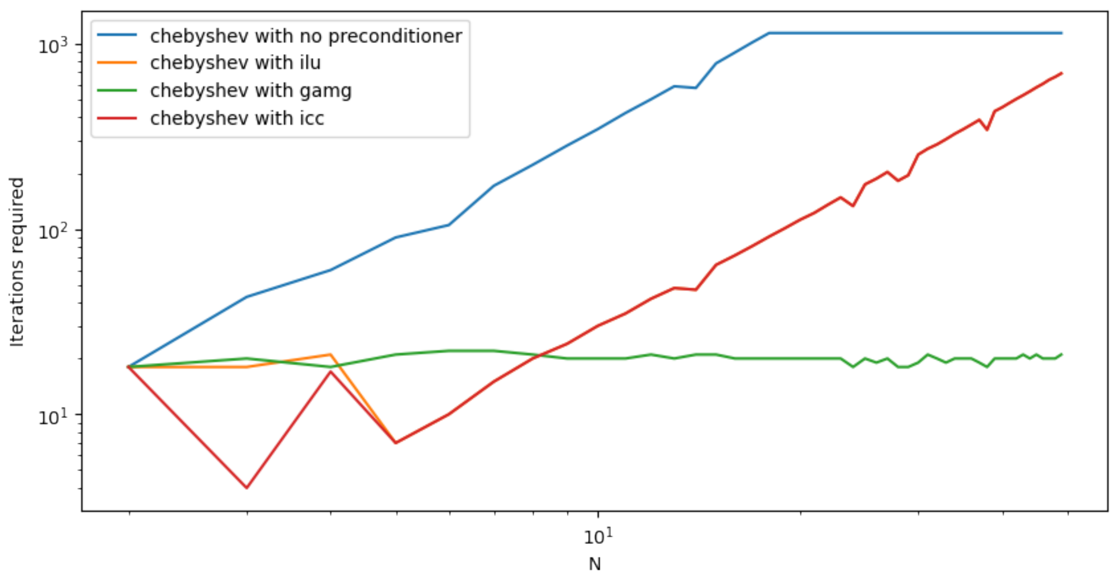
[42]:
```
chebyshev4_times = []
chebyshev4_iters = []
chebyshev4_residuals = []

for N in Ns:

    #Setup our solver
    ksp = PETSc.KSP().create()
    ksp.setType("chebyshev")

    A, f = helmholtz_fe(N)
    #Convert matrix and vector to PETSc
    A = PETSc.Mat().createAIJ(size=A.shape,csr=(A.indptr, A.indices,A.data))
    b = A.createVecLeft()
    b.array[:] = f

    #Vector to solve
    x = A.createVecRight()

    ksp.setOperators(A)
    ksp.setConvergenceHistory()
    ksp.getPC().setType("icc")

    start = perf_counter()

    #Solve the system
    ksp.solve(b, x)
    chebyshev4_times.append(perf_counter() - start)

    #Get convergence info
    res = ksp.getConvergenceHistory()
    chebyshev4_iters.append(len(res))
    chebyshev4_residuals.append(res[-1])
```

[43]:
```
fig, ax = plt.subplots(figsize=(10, 5),dpi=100)
ax.plot(Ns, chebyshev1_times, label = "chebyshev with no preconditioner")
ax.plot(Ns, chebyshev2_times, label = "chebyshev with ilu")
ax.plot(Ns, chebyshev3_times, label = "chebyshev with gamg")
ax.plot(Ns, chebyshev4_times, label = "chebyshev with icc")
ax.set_xlabel("N")
ax.set_ylabel("Time")
ax.set_xscale("log")
ax.set_yscale("log")
ax.legend();
```



For chebyshev, using preconditioning improves the solver run time. However, icc and ilu begin to spike in run time as the matrix size grows. Eventually, gamg becomes the best preconditioner as it grows at a slower rate than the other methods.

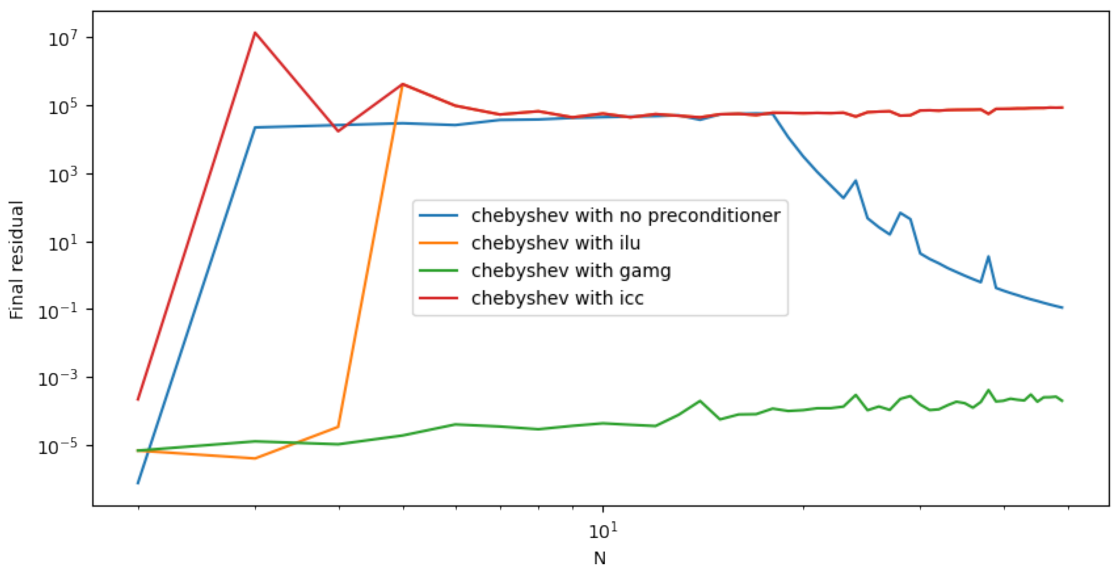
[44]:
```python
fig, ax = plt.subplots(figsize=(10, 5),dpi=100)
ax.plot(Ns, chebyshev1_iters, label = "chebyshev with no preconditioner")
ax.plot(Ns, chebyshev2_iters, label = "chebyshev with ilu")
ax.plot(Ns, chebyshev3_iters, label = "chebyshev with gamg")
ax.plot(Ns, chebyshev4_iters, label = "chebyshev with icc")
ax.set_xlabel("N")
ax.set_ylabel("Iterations required")
ax.set_xscale("log")
ax.set_yscale("log")
ax.legend();
```



It looks like preconditioning reduces the number of required iterations to begin with. However, as the matrix size grows this helps less when it comes to ilu and icc. gamg once again greatly reduces the number of required iterations as the preconditioning step appears to have already almost solved the system.

Let us once again check that all of these solutions converge:

[45]:
```python
fig, ax = plt.subplots(figsize=(10, 5),dpi=100)
ax.plot(Ns, chebyshev1_residuals, label = "chebyshev with no preconditioner")
ax.plot(Ns, chebyshev2_residuals, label = "chebyshev with ilu")
ax.plot(Ns, chebyshev3_residuals, label = "chebyshev with gamg")
ax.plot(Ns, chebyshev4_residuals, label = "chebyshev with icc")
ax.set_xlabel("N")
ax.set_ylabel("Final residual")
ax.set_xscale("log")
ax.set_yscale("log")
ax.legend();
```



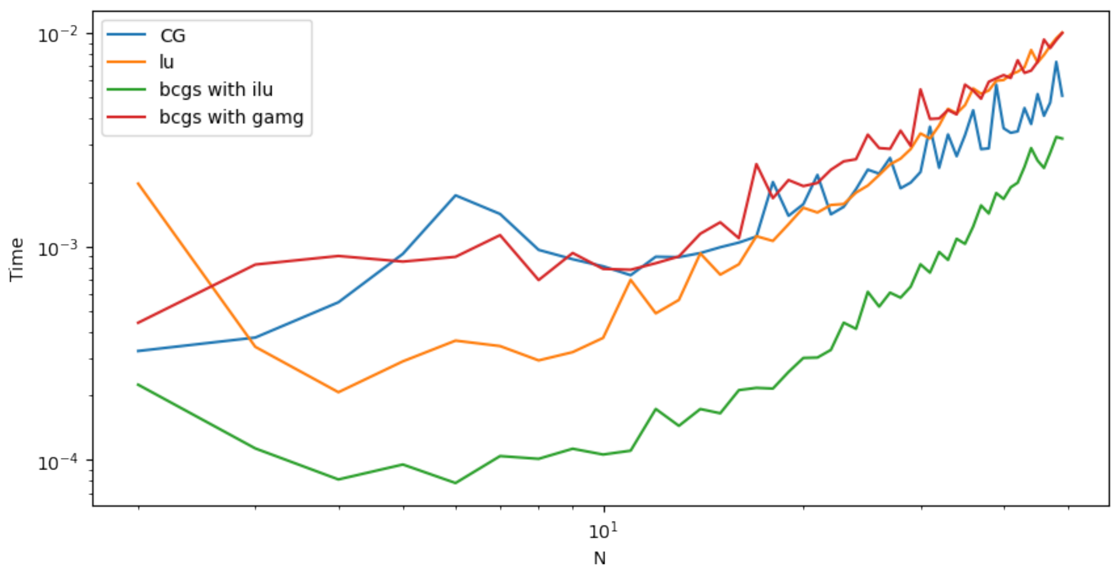
It looks like all the methods, except using gamg as a preconditioner, fail to converge to a small residual. This may be due to assumptions required by the chebyshev solver which this problem does not meet. gamg most likely works because the preconditioning step essential solves the system before it is passed into the iterative solver.

Based on your experiments, **pick a solver** (and a preconditioner if it improves the solver) that you think is most appropriate to solve this matrix-vector problem. **Explain, making use of the data from your experiments, why this is the best solver for this problem**.

1.3.3 My choice in solver

To decide on the solver I will use in the next section, we will compare the best performing direct solver, scipy solver and PETSc solver with preconditioning:

[46]:
```
fig, ax = plt.subplots(figsize=(10, 5),dpi=100)
ax.plot(Ns, cg_times, label = "CG")
ax.plot(Ns, splu_times, label = "lu")
ax.plot(Ns, bcgs2_times, label = "bcgs with ilu")
ax.plot(Ns, bcgs3_times, label = "bcgs with gamg")
ax.set_xlabel("N")
ax.set_ylabel("Time")
ax.set_xscale("log")
ax.set_yscale("log")
ax.legend();
```



[47]:
```
print(f"Difference between spsolve and cg solution: {np.
  ↪mean(abs(spsolve_sol-cg_sol))}")
print(f"Difference between spsolve and PETSc.bcgs + ilu solution: {np.
  ↪mean(abs(spsolve_sol-bcgs_ilu_sol))}")
print(f"Difference between spsolve and PETSc.bcgs + gamg solution: {np.
  ↪mean(abs(spsolve_sol-bcgs_gamg_sol))}")
```

```
Difference between spsolve and cg solution: 4.319592881767775e-06
Difference between spsolve and PETSc.bcgs + ilu solution: 3.438139965728656e-05
Difference between spsolve and PETSc.bcgs + gamg solution:
1.9771937987293503e-07
```

Solver	type	Pros	Cons
`scipy.cg`	iterative	Appears the be the second fastest method - also appears to be growing at a slower rate than the fastest method so may be better for large matrices	The convergence process is not very smooth - isn't the fastest
`scipy.splu`	direct	Essentially no error as it is a direct method	Slower than the iterative methods and appears to be growing at a faster rate than cg for the larger matrices

Solver	type	Pros	Cons
`PETSc.bcgs + ilu`	iterative with preconditioning	The fastest method	The run time and number of required iterations grows quickly as matrix size increases
`PETSc.bcgs + gamg`	iterative with preconditioning	Greatly reduces the number of iterations required by the iterative solver	Appears to be the slowest for smaller matrices

I'm going to use cg with no preconditioner for the next section. It performed the best out of the scipy solvers I used as the matrix size grew larger and appears to be growing in run time at a slower rate than bcgs with ilu. Testing it's solutions with the direct solvers also suggested that it was giving fairly precise answers.

1.4 Part 4: increasing N

In this section, you are going to use the solver you picked in part 3 to compute the solution for larger values of N.

The problem we have been solving in this assignment has the exact solution $u_{\text{exact}} = \sin(3x + 4y)$. A measure of the error of an approximate solution u_h can be computed using

$$\sum_{i=0}^{N^2-1} h^2 \left| u_{\text{exact}}(\mathbf{m}_i) - u_h(\mathbf{m}_i) \right|,$$

where $\mathbf{m}_i$ is the midpoint of the ith square in the finite element mesh: the value of u_h at this midpoint will be the mean of the values at the four corners of the square.

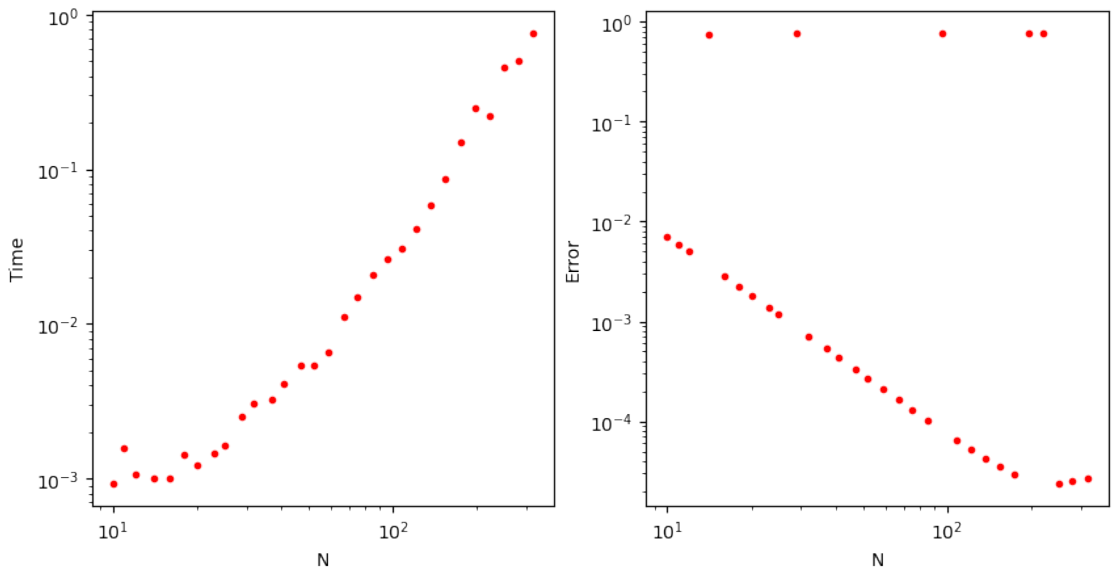
For a range of values of N from small to large, **compute the solution to the matrix-vector problem**. **Measure the time taken to compute this solution**, and **compute the error of the solution**. **Make plots showing the time taken and error as N is increased**.

[48]:
```
def get_error(sol, N):

    h = 1/N

    #Evaluate the boundary points
    ticks= np.linspace(0, 1, N+1)
    X, Y = np.meshgrid(ticks, ticks)
    u_exact = g(X, Y) #This evaluates g for the full grid

    u_h = g(X, Y) #keep the boundary points for the solved solution
    u_h[1:N, 1:N] = sol.reshape((N-1, N-1)) #Replace the interior points with␣
↪the matrix vector solutions
```

```
    err = 0
    for i in range(0,(N)**2):
        p_i = (i//(N), i%(N))

        u_exact_mi = np.mean(u_exact[p_i[0]:p_i[0]+2,p_i[1]:p_i[1]+2])
        u_h_mi = np.mean(u_h[p_i[0]:p_i[0]+2,p_i[1]:p_i[1]+2])

        err += h**2*abs(u_exact_mi-u_h_mi)

    return err
```

[49]:
```
errors = []
times = []
Ns = np.logspace(1,2.5,30, dtype=int)

for N in Ns:

    #Setup our solver
    A, f = helmholtz_fe(N)

    #Solve the system
    start = perf_counter()
    sol = cg(A,f)[0]
    times.append(perf_counter() - start)

    #Check the error
    errors.append(get_error(sol, N))
```

[50]:
```
fig, ax = plt.subplots(1,2,figsize=(10, 5),dpi=100)

ax[0].scatter(Ns, times, marker='.',c='r')
ax[0].set_xlabel("N")
ax[0].set_ylabel("Time")
ax[0].set_xscale("log")
ax[0].set_yscale("log")

ax[1].scatter(Ns, errors, marker='.',c='r')
ax[1].set_xlabel("N")
ax[1].set_ylabel("Error")
ax[1].set_xscale("log")
ax[1].set_yscale("log");
```



Using your plots, **estimate the complexity of the solver you are using** (ie is it $\mathcal{O}(N)$? Is it $\mathcal{O}(N^2)$?), and **estimate the order of convergence of your solution** (your error should decrease like $\mathcal{O}(N^{-\alpha})$ for some $\alpha > 0$). Briefly (1-2 sentences) **comment on how you have made these estimates of the complexity and order.**

Both the time complexity and the error convergence appear to follow a polynomial. To find the exponent of this polynomial, α, I will fit a function, AN^{α}, to the time and error data using the curve fit function from scipy:

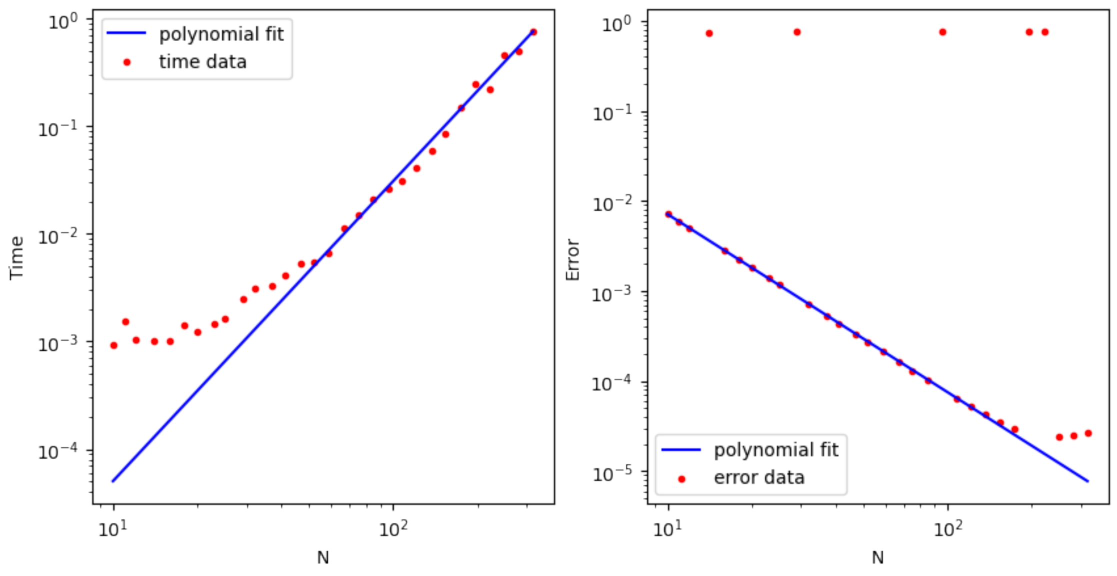
[51]:
```
from scipy.optimize import curve_fit
from scipy import stats
```

[52]:
```
#Function to be fit
def func(x,A,alpha):

    return A*x**(alpha)
```

[53]:
```
#Fit the exponential to the times
popt_times = curve_fit(func, Ns, times)[0]

#Fit the exponential to the errors - I've filtered out the outliers using a z␣
↪score filter
popt_err = curve_fit(func, Ns[np.abs(stats.zscore(errors)) < 1]
                     ,np.array(errors)[np.abs(stats.zscore(errors)) < 1] )[0]
```

[54]:
```
#Plot the fits

fig, ax = plt.subplots(1,2,figsize=(10, 5),dpi=100)
```

```
ax[0].scatter(Ns, times, label = "time data", marker='.',c='r')
ax[0].plot(Ns, func(Ns, *popt_times), label = "polynomial fit", c='b')
ax[0].set_xlabel("N")
ax[0].set_ylabel("Time")
ax[0].set_xscale("log")
ax[0].set_yscale("log")
ax[0].legend()

ax[1].scatter(Ns, errors, label = "error data", marker='.',c='r')
ax[1].plot(Ns, func(Ns, *popt_err), label = "polynomial fit", c='b')
ax[1].set_xlabel("N")
ax[1].set_ylabel("Error")
ax[1].set_xscale("log")
ax[1].set_yscale("log")
ax[1].legend();
```



These both appear to have produced good fits. We will now check the value of α obtained for both cases:

[55]:
```
print(f"Time complexity : O(N^{popt_times[1]})")
print(f"Order of convergence : alpha = {-popt_err[1]}")
```

```
Time complexity : O(N^2.7837108394362624)
Order of convergence : alpha = 1.978020040193089
```

These fits imply that the time complexity of the solution is roughly $\mathcal{O}(N^{2.8})$, while the convergence of the problem is $\mathcal{O}(N^{-2})$. However, it looks like the cg error begins to grow slightly as the matrix size approaches 10^3, this may mean it is not the best choice in solver for really large matrices.

Quickly looking at a direct solver instead:

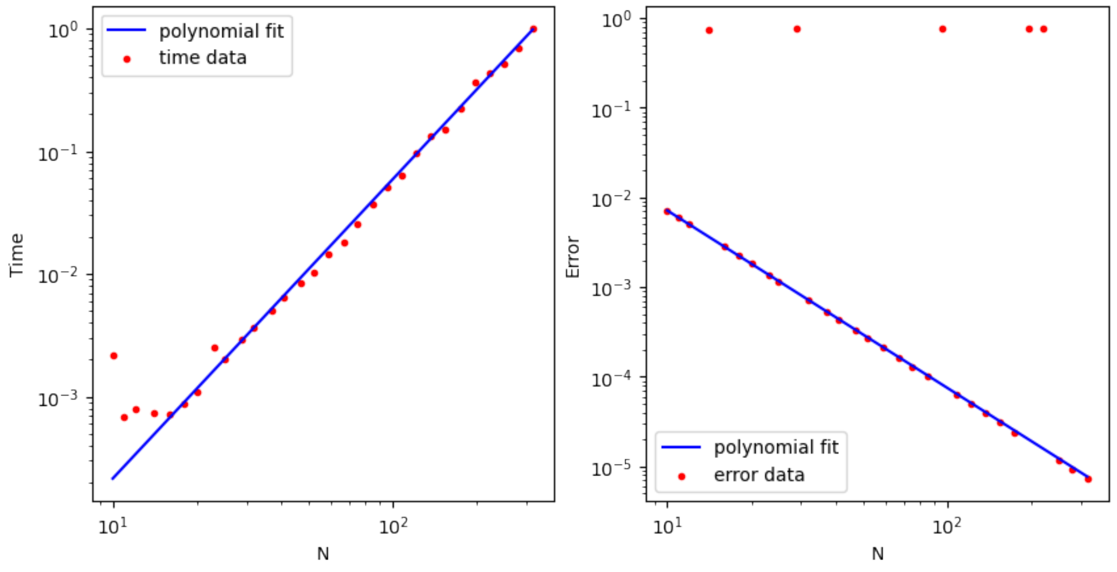
[56]:
```python
errors = []
times = []
Ns = np.logspace(1,2.5,30, dtype=int)

for N in Ns:

    #Setup our solver
    A, f = helmholtz_fe(N)

    #Solve the system
    start = perf_counter()
    sol = splu(A).solve(f)
    times.append(perf_counter() - start)

    #Check the error
    errors.append(get_error(sol, N))
```

```
/Users/james/opt/anaconda3/lib/python3.8/site-
packages/scipy/sparse/linalg/_dsolve/linsolve.py:347: SparseEfficiencyWarning:
splu converted its input to CSC format
  warn('splu converted its input to CSC format', SparseEfficiencyWarning)
```

[57]:
```python
#Fit the exponential to the times
popt_times = curve_fit(func, Ns, times)[0]

#Fit the exponential to the errors - I've filtered out the outliers using a z␣
↪score filter
popt_err = curve_fit(func, Ns[np.abs(stats.zscore(errors)) < 1]
                     ,np.array(errors)[np.abs(stats.zscore(errors)) < 1] )[0]
```

[58]:
```python
#Plot the fits

fig, ax = plt.subplots(1,2,figsize=(10, 5),dpi=100)

ax[0].scatter(Ns, times, label = "time data", marker='.',c='r')
ax[0].plot(Ns, func(Ns, *popt_times), label = "polynomial fit", c='b')
ax[0].set_xlabel("N")
ax[0].set_ylabel("Time")
ax[0].set_xscale("log")
ax[0].set_yscale("log")
ax[0].legend()

ax[1].scatter(Ns, errors, label = "error data", marker='.',c='r')
ax[1].plot(Ns, func(Ns, *popt_err), label = "polynomial fit", c='b')
ax[1].set_xlabel("N")
ax[1].set_ylabel("Error")
```

```
ax[1].set_xscale("log")
ax[1].set_yscale("log")
ax[1].legend();
```

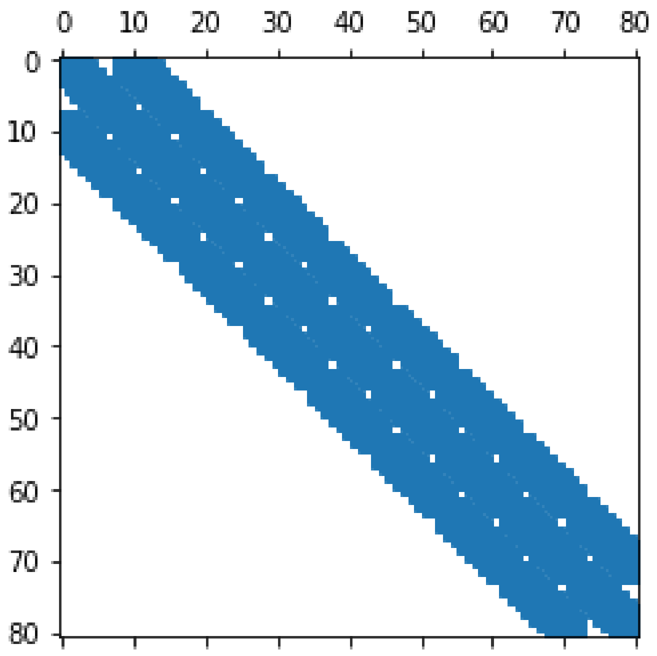


```
[59]: print(f"Time complexity : O(N^{popt_times[1]})")
      print(f"Order of convergence : alpha = {-popt_err[1]}")
```

```
Time complexity : O(N^2.440626033533478)
Order of convergence : alpha = 1.9786630743751648
```

LU has a similar time complexity and error decrease, but doesn't experience the increase in error for large matrices. **Quick note** - looking at the structure of our matrix. It appears to be banded which may be why LU performs well for large matrix sizes

```
[60]: A, f = helmholtz_fe(10)
      plt.spy(A);
```



1.5 Part 5: parallelisation

In this section, we will consider how your solution method could be parallelised; you do not need, however, to implement a parallel version of your solution method.

Comment on how your solution method could be parallelised. Which parts (if any) would be trivial to parallelise? Which parts (if any) would be difficult to parallelise? By how much would you expect parallelisation to speed up your solution method?

1.5.1 Creating the matrix and vector

The process of creating the finite element matrix and vector should be a fairly simple step to parallelize. Each matrix and vector element is obtained independently and so could be computed on a different core. Depending of the number of cores available (n), we could compute n matrix and vector elements simultaneously. Some of the calculations required for different matrix and vector elements are the same, this fact could also be used to cut down on the computation time of creating the matrix and vector.

1.5.2 Solving the matrix vector problem

Iterative solvers These may be not so simple to parallelise as each iteration relies on the solution from the iteration before. However, certain calculations within each iteration may be computed in parallel if values used in one calculation aren't the result of a different calculation within the iteration. For example cg has the following form:

$$d_0 = r_0 = b - Ax_0$$

$$\alpha_i = \frac{r_i^T r_i}{d_i^T A d_i}$$
$$x_{i+1} = x_i + \alpha_i d_i$$
$$r_{i+1} = r_i - \alpha_i A d_i$$
$$\beta_{i+1} = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i}$$
$$d_{i+1} = r_{i+1} + \beta_{i+1} d_i$$

The steps for x_{i+1} and r_{i+1} don't rely on one another and so can be parallelised. If an iterative solver had a great number of such calculations, then these may all be run concurrently. Furthermore, the additions, subtractions and multiplications within each calculation are operations performed on matrices and vectors - all of which can be run in parallel (to some extent) - e.g. adding two large vector together can be done in a single step given that a lot of cores are available.

Direct solver/preconditioners Similarly to iterative solvers, parallelising these methods depends on the type of direct solver or preconditioner being used. Methods which contain lots of independent calculations are easier to parallelise as these calculations can be performed concurrently. Calculations which involve matrix additions and multiplications can also be parallelised. Methods such as LU alter the matrices they are deal with at each calculation and therefore these steps cannot be run in parallel - however, steps such as subtracting a given row of a matrix from all rows below it may be run as one large operation.

Therefore, the simplest way to speed up any solver using parallelisation is to apply the technique to the matrix and vector operations. As the size of the matrices grow, the number of computational steps required for each operation will remain the same (for addition and subtraction) up to the limit of the number of cores used.